A Comparative Study of Continuous Integration and Continuous Delivery Tools in a DevSecOps Process for Mobile Application Development in Small Organizations

Kamchai Boonsri
College of Arts, Media and Technology
Chiang Mai University
Chiang Mai, Thailand
kamchai_b@cmu.ac.th

Chartchai Doungsa-ard
College of Arts, Media and Technology
Chiang Mai University
Chiang Mai, Thailand
chartchai.d@cmu.ac.th

Abstract*—This study compares the performance of three Continuous Integration and Continuous Delivery (CI/CD) tools—Jenkins, GitHub Actions, and GitLab CI/CD—within a DevSecOps pipeline for mobile application development in the context of small organizations. Most prior research has focused on general or web-based software, so the mobile pipeline remains under-studied. The authors designed a five-stage DevSecOps pipeline and built two prototype applications with the Flutter framework: a small Task Management application and a medium-sized Home Repair and Maintenance Services application. Each tool ran the complete pipeline 20 times for each application size, which gives six experimental conditions and a total sample of 120 pipeline runs (n = 3 tools × 2 application sizes × 20 runs per condition = 120). Pipeline duration and system resource usage (CPU and memory) were measured automatically. Jenkins delivered the fastest pipeline (median 127 s for the small app and 156 s for the medium app), GitLab CI/CD ranked second and was the most consistent tool, and GitHub Actions had the longest and most variable execution time. CPU usage did not differ significantly across tools, but Jenkins consumed the most memory and GitHub Actions the least. The findings reveal a clear trade-off between speed and resource use, and provide practical guidance: Jenkins for speed-driven teams, GitHub Actions for teams with limited staff and infrastructure, and GitLab CI/CD for teams that need predictable delivery.

***Keywords*—DevSecOps; CI/CD; mobile application; Flutter; Jenkins; GitHub Actions; GitLab CI/CD;**

I. Introduction

Mobile application development is a complex process. Development teams must design the user interface, implement the functional requirements, achieve broad test coverage, and verify the behavior of the application on both Android and iOS. In small organizations, where staff and budget are limited, this work has to be planned carefully so that delivery to users remains fast and reliable. Frequent releases have become a key competitive factor in the mobile industry: users expect new features and quick bug fixes, and a slow release process directly harms user satisfaction and damages the reputation of the organization.

In many small teams, mobile delivery is still carried out manually. Developers compile the application for each platform and then upload the artifacts to the application stores by hand. This manual workflow is slow, complex, and prone to human error. To address these issues, the DevOps culture promotes close cooperation between the development and operations teams and applies automation across coding, testing, integration, and delivery. Because security has also become critical, this idea has evolved into DevSecOps, in which security checks are embedded in every stage of the pipeline. DevSecOps helps teams detect weaknesses early, reduces risk, and increases user trust in the product.

The effectiveness of a DevSecOps process depends on the tools that automate it. In this study, the right tool means a CI/CD tool that can execute the complete mobile pipeline—build, automated testing, static code analysis, security scanning, and delivery to the application stores—reliably, and within the staffing, infrastructure, and budget limits of a small organization. Jenkins, GitHub Actions, and GitLab CI/CD are the three most widely adopted candidates, and they differ in configuration effort, hosting model, and resource consumption. Earlier comparative studies have focused on microservices in the cloud [1] or on web applications [2]. More recent work on mobile projects has examined how pipelines are configured rather than how the tools perform: an empirical study of 2,557 open-source Android applications [3] reports that GitHub Actions is used by 65% of the projects and GitLab CI/CD by only 3%, that fewer than 9% of the projects automate deployment at all, and that only 11% publish directly to Google Play, with most of the remainder relying on Fastlane. That study therefore describes current practice in mobile pipelines, but it does not measure the performance of the tools themselves. The performance of these tools in mobile pipelines, which additionally require platform-specific build commands, code signing, and store delivery, has not yet been studied in depth. This paper addresses that gap by comparing the three tools on a single DevSecOps pipeline that targets mobile applications in small organizations. The work has two objectives: (1) to compare the performance of the three CI/CD tools, and (2) to design a DevSecOps pipeline that fits the constraints of small organizations.

II. Literature Review

A. Mobile Application Size

Mobile applications can be grouped by complexity. Following the classification used by Deviniti [4], a small application has up to five screens, a linear flow, no external API, and basic graphics. A medium application has 5–15 screens, a feature-based structure, several external APIs, and standard graphics. A large application has more than 15 screens, multi-level navigation, advanced security, and many integrations. The 5–15 range for medium applications is taken directly from this source classification, and its lower bound deliberately meets the upper bound of the small class rather than starting at six. The reason is that screen count alone does not separate the two classes. An application with five screens is small only when it also has a linear flow, no external API, and basic graphics; the same five screens define a medium application as soon as the flow branches by feature, several external APIs are called, and standard graphics are used. Screen count therefore acts as an upper limit for the small

class and as only one of four criteria for the medium class, and the two ranges overlap at five by design. The study uses small and medium applications because these are the most common types built by small organizations and because their difference in size is large enough to reveal differences between the CI/CD tools.

B. Cross-Platform Development with Flutter

Mobile applications can be built natively, using Kotlin or Java for Android [5] and Swift or Objective-C for iOS [6], or with cross-platform frameworks such as Flutter [7] and React Native [8]. Cross-platform development allows a single codebase to target multiple operating systems, which lowers cost and shortens delivery time. The Thai Developer Salary Report 2023 [9] reported that Flutter is the most popular mobile framework in Thailand, used by about 66 percent of mobile developers. The authors selected Flutter for both prototypes because it suits the resource limits of small organizations and reduces duplicated work between platforms.

C. DevSecOps and Performance Indicators

DevSecOps is a modern software approach that combines DevOps with continuous security [10]. It promotes shared responsibility for security across development, operations, and quality teams. According to industry data, the average cost of a data breach in 2022 was about USD 4.35 million [10], which has pushed many organizations to integrate security earlier in the lifecycle. The DevOps Research and Assessment (DORA) program [11] defines four software delivery indicators. Lead time is the elapsed time from the moment a code change is committed to the moment that change runs in production; it is measured by time-stamping the commit and the successful deployment and taking the difference. Deployment frequency is how often the team releases code to production; it is measured by counting successful deployments per unit of time, for example per day or per week. Failure rate is the proportion of deployments that cause a failure in production and require a fix; it is measured as the number of failed deployments divided by the total number of deployments over the same period. Mean time to recovery is the average time needed to restore service after such a failure; it is measured from incident records as the interval between the start of the failure and the restoration of normal operation. This study uses lead time as the main time indicator because it reflects the full pipeline from code change to delivery, and because it can be measured directly in a controlled experiment: the CI Metrics Collector records the wall-clock duration from the triggering commit to the completion of store delivery, broken down by pipeline stage. The remaining three indicators require a production release history over an extended observation window and therefore fall outside the scope of this experiment. The authors also add system resource utilization (CPU and memory), which is not part of the DORA framework but is important for small organizations with limited infrastructure budgets; it is measured by sampling CPU and memory consumption every five seconds while the pipeline runs.

III. METHODOLOGY

This is an experimental study that compares three CI/CD tools under one DevSecOps pipeline. The independent variables are the CI/CD tool (Jenkins, GitHub Actions and GitLab CI/CD) and the application size (small and medium). The dependent variables are pipeline execution time and system resource usage. The framework of the study is shown conceptually in Fig. 1.

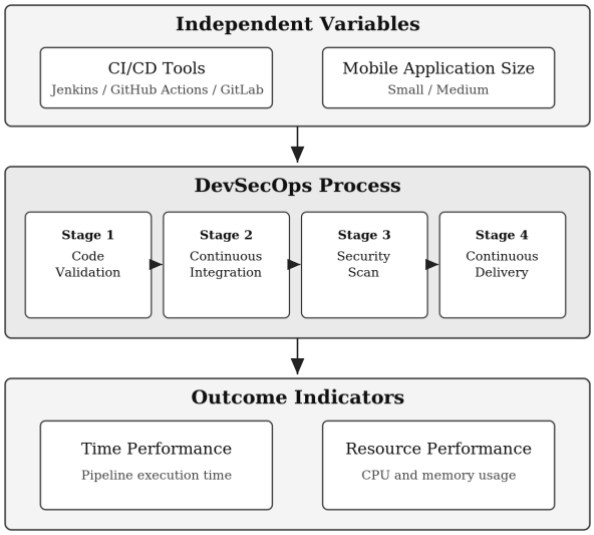

Fig. 1. Conceptual research framework of the comparative study.

A. Designed DevSecOps Pipeline

The authors designed a five-stage automated pipeline, following common CI/CD pipeline patterns [12]. Stage 1 is Development, in which developers commit code on a feature branch and open a pull request. Stage 2 is Code Validation, which runs unit tests and SonarQube [13] static analysis. Stage 3 is Continuous Integration, in which Flutter is used to build the .ipa file for iOS and the .aab file for Android. Stage 4 is Security Scan, in which the Mobile Security Framework (MobSF) [14] checks the application; MobSF classifies its security score into four risk levels, namely 0–29 (critical risk), 30–39 (high risk), 40–59 (medium risk), and 60–100 (low risk), and the pass criterion adopted here is a score of 40 or above, that is, medium risk or better. This threshold reuses the boundary defined by the tool itself instead of a value invented by the authors, so the criterion is traceable to its source and matches the risk level shown in the MobSF report. Stage 5 is Continuous Delivery, in which Fastlane [15] uploads the artifacts to Google Play Console (Internal Testing) and to App Store Connect (TestFlight). Each stage has a pass rule, and a failure stops the pipeline at once and notifies the development team. The overall pipeline is illustrated in Fig. 2.

B. Prototype Applications

Two prototypes were built with Flutter 3.24.5 to cover both ends of the small-organization workload. The small application is a Task Management app with five screens that supports add, edit, delete, and completion marks; it has a linear flow, no external API, and basic graphics. The medium application is a Home Repair and Maintenance Services app that lies within the 5–15 screen band of the medium class, with user registration, booking, and a booking history with service status; it has a feature-based structure and calls several external APIs. Both prototypes follow Clean Architecture and include unit tests, which makes the pipeline runs comparable across tools.

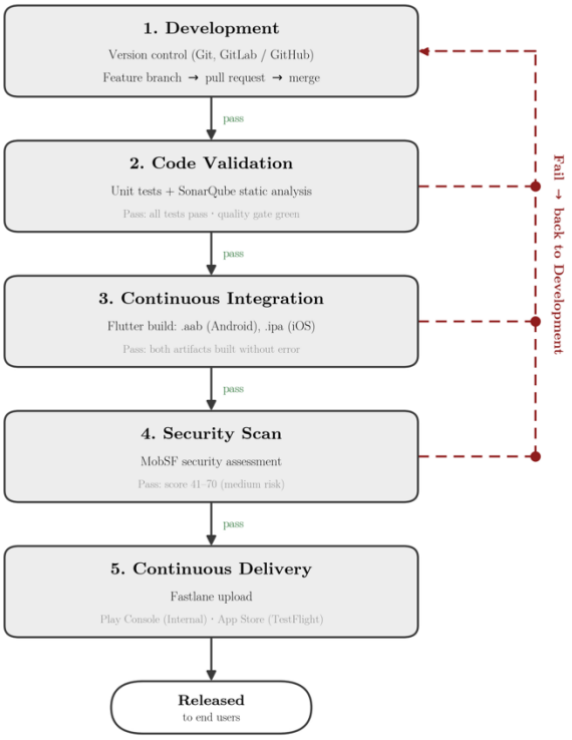

Fig. 2. The designed five-stage DevSecOps pipeline.

C. Experimental Setup and Tools

All experiments ran on a Mac mini with Apple M4, 16 GB of memory, and 512 GB of storage on macOS 15.1.1. The CI tools used were Jenkins 2.479.2, GitHub Actions, and GitLab CI/CD. Quality and security tools were SonarQube 10.8.0 and MobSF 4.1.1. Fastlane 2.225.0 handled delivery to the stores. SonarQube, MobSF, and Jenkins were run in containers managed by Docker [16] 27.4.0 and Docker Compose 2.31.0. The GitLab Runner and Fastlane were installed natively through Homebrew. GitHub Actions used a self-hosted runner that pointed to the same Mac mini, so the hardware was identical for every tool.

D. Test Procedure and Data Collection

A CI Job Tracker written in Python drove the experiment from end to end. For each run, the tracker waited for a random delay of three to five minutes in order to simulate a realistic commit pattern, and then pushed a small change to the branch that triggered the selected tool. While the pipeline was running, a CI Metrics Collector recorded the start and end time of every stage and sampled CPU and memory usage every five seconds. After the pipeline had finished, the collected data were validated and stored in PostgreSQL [17] in two related tables: jobs, which holds the configuration, and job_runs, which holds the execution data including a JSON metrics field. The procedure was repeated until 20 complete runs had been recorded for each condition, which gives a total of 120 runs (3 tools × 2 application sizes × 20 runs per condition). Here, one run denotes one complete execution of the five-stage pipeline, from the triggering commit to the completion of store delivery, and only runs with complete measurements for every indicator were counted towards the 20, so that all six conditions contribute an equal number of observations.

E. Statistical Analysis

The collected data were first checked for completeness and for outliers. Outliers were detected with the 1.5×IQR rule proposed by Tukey [18], which flags any value below Q1 − 1.5×IQR or above Q3 + 1.5×IQR, where Q1 and Q3 are the first and third quartiles and IQR = Q3 − Q1. The multiplier 1.5 is Tukey's conventional constant and is not a value chosen by the authors: for normally distributed data the resulting fences fall at approximately ±2.7 standard deviations from the mean, so less than 1% of observations are flagged by chance. A smaller multiplier would mark ordinary variation as abnormal, while a larger one would hide genuine extreme values. The flagged values were nevertheless kept in the dataset, because in tool benchmarking they usually reflect real tool behavior, such as an unusually long queue wait, rather than measurement error, and removing them would make the results look better than the tools actually perform. Descriptive statistics (mean, median, SD, minimum, maximum) were then computed. The Shapiro–Wilk test was used to check normality. Because most groups were not normally distributed, the authors chose nonparametric tests: the Kruskal–Wallis test [19] for the overall comparison and the Dwass-Steel-Critchlow-Fligner (DSCF) test [20] for pairwise comparisons. The significance level was set at $\alpha = 0.05$ in all tests. This value is the standard threshold in experimental research and means that the authors accept a risk of at most 5% of concluding that the tools differ when in fact they do not, which is a Type I error; it was fixed before the analysis so that the decision criterion could not be adjusted after the results had been seen. Effect size was reported with epsilon-squared (ε^2), for which 0.01–0.06 indicates a small effect, 0.06–0.14 a medium effect, and above 0.14 a large effect.

IV. Results

Before the results are presented, the criteria used to interpret them are stated so that every indicator in this section is read in the same way. All tests are judged against a significance level of $\alpha = 0.05$, which was fixed in advance of the analysis so that the decision threshold could not be adjusted once the results were known. The p-value is the probability of obtaining a difference at least as large as the one measured if the three tools in fact performed identically. A p-value below .05 therefore means that such a difference would arise by chance less than 5% of the time, so the difference is reported as statistically significant and the tools are ranked on that indicator; the notation $p < .001$ marks the strongest case, in which that probability is below 0.1%. A p-value of .05 or above means that the data do not provide enough evidence to conclude that the tools differ, which is not the same as proving that they are equal; in that case the measured values are reported for reference but no ranking is made. Effect size is reported together with the p-value, because the p-value answers whether a difference is reliable while epsilon-squared answers how large that difference is.

A. Time Performance

Jenkins produced the fastest pipeline at both application sizes, followed by GitLab CI/CD, with GitHub Actions the slowest (Table 1). GitLab CI/CD was also the most consistent tool and GitHub Actions the most variable, as shown by the interquartile range in Table 2. The Kruskal–Wallis test confirmed significant differences between the three tools in nearly all time stages ($p < .001$), with large to very large effect

sizes ($\varepsilon^2 = 0.572–0.901$). The only exception was Continuous Delivery, where Jenkins and GitLab CI/CD did not differ significantly ($p = 0.864$ for the small application and $p = 0.228$ for the medium application), because this stage is bounded by the response time of the Apple and Google servers rather than by the tool.

TABLE 1. MEDIAN PIPELINE TIME (SECONDS)

App Size	Jenkins	GitLab CI/CD	GitHub Actions
Small	127	259	423
Medium	156	291	436

TABLE 2. PIPELINE TIME VARIABILITY (IQR, SECONDS)

App Size	Jenkins	GitLab CI/CD	GitHub Actions
Small (IQR)	18.0	11.3	68.5
Medium (IQR)	21.0	11.0	161.0

B. Resource Utilization

CPU utilization showed no significant difference between the three tools at either size ($p = 0.234$ and $p = 0.414$), because the CPU profile is driven by the Flutter build, the unit tests, and the SonarQube and MobSF scans rather than by the tool. Memory usage, however, differed significantly across all pairs ($p < .05$): Jenkins used the most, GitHub Actions the least, and GitLab CI/CD fell between them (Table 3).

TABLE 3. MEDIAN MEMORY USAGE (MEGABYTES)

App Size	Jenkins	GitLab CI/CD	GitHub Actions
Small	10,136	9,404	9,218
Medium	10,546	9,596	9,293

C. Effect of Application Size

The Code Validation, Continuous Integration, and Continuous Delivery stages were significantly slower on the medium application ($p < .05$), as expected from its larger codebase, more dependencies, and additional API calls. The Security Scan stage, however, did not differ significantly between sizes ($p > .05$): MobSF relies mainly on pattern matching and inspection of configuration files such as AndroidManifest.xml and Info.plist, so its cost stays stable regardless of source-code size.

V. DISCUSSION

The results show a trade-off between speed and resource use. Jenkins is the fastest tool because it is self-hosted and uses the hardware directly, with no shared layer, which removes external delay. This finding agrees with Kuncara, Kusumo and Adrian (2024) [2], who reported that Jenkins gives shorter deployment time than other tools. The price is higher memory use, because Jenkins keeps more data in cache and runs more steps in parallel.

GitHub Actions is the slowest and the most variable tool. As a cloud-hosted service, every run starts from a fresh environment, which adds queue time and time spent downloading dependencies and artifacts. These costs depend on network load and on the number of jobs running on the platform at the same time, so the variance is large. The advantage is the lowest memory footprint, because the platform applies strict resource limits to share infrastructure across many users. A large part of this overhead is structural: GitHub Actions and GitLab CI/CD run each stage as a separate job and exchange build artifacts through the platform between stages, whereas Jenkins keeps all stages in one local workspace and passes artifacts on local disk.

GitLab CI/CD takes the middle position. Its hybrid model, which uses local runners that still talk to a central server, gives a good balance between speed and resource use. Most importantly, it is the most consistent tool: a low IQR means that delivery time is predictable, which is a strong advantage when small organizations need to commit to a release schedule.

Application size affects the build-related stages but not the security scan. Within the range used in this study (5–15 screens), MobSF time is almost constant. This suggests that, in small and medium projects, security cost is not a barrier to adopting DevSecOps, because the time it adds is fixed and small.

Small organizations often face two pressures at the same time: short release cycles and tight budgets for staff and infrastructure. The trade-off observed here means that no single tool dominates on every metric, so the choice has to follow the local context. When the team has only one or two people who can run a server, a managed cloud option such as GitHub Actions removes operational work, even if each pipeline run is slower. When release predictability is more important than absolute speed (for example, when releases are tied to a sprint demo), GitLab CI/CD has a clear advantage because of its low IQR. When build time is a direct part of customer experience, for example for an internal beta channel that ships several times a day, Jenkins is justified, but the team must also plan for the higher memory cost and for keeping the runner up to date.

To make this guidance easier to apply in practice, Table 4 maps common situations of small organizations to a recommended tool, together with a short reason. The table is meant as a starting point; teams should still combine it with local factors such as team skill, system compatibility, customisation needs, and the organization's security policy before making a final decision.

TABLE 4. TOOL SELECTION GUIDE FOR SMALL ORGANIZATIONS

Organization Situation	Tool	Reason and Notes
Experienced DevOps team with own server, looking for the highest performance	**Jenkins**	Fastest pipeline (127–156 s) and consistent runs, but uses the most memory (10,136–10,546 MB) and needs strong setup and maintenance skills.
Small team, tight budget, limited server resources, GitHub as main platform	**GitHub Actions**	Lowest memory usage (9,218–9,293 MB) and seamless GitHub integration with no infrastructure to manage, but the slowest tool (423–436 s) with high variability.
Team that needs flexibility, expects growth, and wants a balance of speed and effort	**GitLab CI/CD**	Mid-range time (259–291 s) with the highest consistency. Supports both self-hosted and cloud runners; suitable for organizations that plan to scale.
Start-up or team without DevOps experience, needs to start fast	**GitHub Actions**	Easy to set up with low entry cost. A good fit for proof-of-concept projects and early experiments.
Strict security requirements or full data control	**Jenkins or GitLab CI/CD**	Both can run fully on-premises, which lets the organization control data and follow strict security policies, but they need investment in infrastructure and people.

Beyond the technical factors above, organizations should also weigh contextual factors such as team skill and experience, compatibility with existing systems, customization needs, and the security policy of the organization. A good decision considers all of these together so that the resulting DevSecOps pipeline truly fits the local context.

VI. Conclusion and Recommendations

This study compared Jenkins, GitHub Actions, and GitLab CI/CD inside a five-stage DevSecOps pipeline for mobile applications in small organizations, using Flutter prototypes of small and medium size. With 120 pipeline runs and nonparametric statistics, the authors found that Jenkins delivered the fastest pipeline at both sizes, GitLab CI/CD was second and the most consistent, and GitHub Actions was the slowest with the highest variance. CPU usage did not differ between tools, while Jenkins used the most memory and GitHub Actions the least. These findings expose a clear trade-off: speed costs memory and predictability.

From these results, the authors propose three context-based recommendations. First, Jenkins is recommended when speed is the top priority and the team has the skills to install and maintain a self-hosted server. Second, GitHub Actions is recommended when staffing or the infrastructure budget is limited, and particularly when the team already hosts its code on GitHub, because the longer run time is offset by the lower memory cost and the simpler setup. Third, GitLab CI/CD is recommended when the organization needs predictable delivery and a balanced use of resources, or when growth is expected and the organization prefers a single platform for the repository, CI/CD, and security review.

This study has three main limitations: all runs were executed on a single Mac mini rather than on Linux or cloud-only infrastructure; the prototypes were Flutter applications rather than native Kotlin or Swift projects; and only two of the four DORA indicators were measured. The future directions below are designed to address these limitations.

A. Future Research Directions

Future work can extend this study in three directions. The first is to add the remaining DORA indicators (deployment frequency, change failure rate, mean time to recovery) over a longer observation window. The second is to compare cloud-hosted runners against self-hosted runners for the same tool in order to isolate the effect of the hosting model. The third is to repeat the study on large applications with more than 15 screens, where dependency graphs and security analysis may behave differently and may change the ranking of the tools.

B. Practical Extensions

Beyond these directions, two practical extensions also seem useful. A cost model that puts the time and memory results in monetary terms would help small organizations justify a tool choice with their finance teams. The cost would include cloud minutes, on-premises hardware depreciation, and engineering hours for maintenance. A second extension is to add a feedback loop from production: linking the pipeline metrics to incident reports would allow teams to check whether a faster pipeline really shortens recovery time, or whether it only shifts work earlier in the lifecycle. Both extensions stay close to the daily concerns of small organizations and could be supported with the dataset and the Python tracker built in this study.

Acknowledgment

The authors would like to thank the College of Arts, Media and Technology, Chiang Mai University, for the academic support and the laboratory resources that made this experiment possible. The authors are also grateful to the examination committee of the Master of Science programme in Software Engineering for the comments and suggestions that improved the accuracy and the completeness of this work, and to the faculty members of the programme for the knowledge and the experience they shared throughout the course of study. Finally, the authors thank their families for their continuous encouragement and support.

References

[1] C. Singh, N. S. Gaba, M. Kaur, and B. Kaur, "Comparison of Different CI/CD Tools Integrated with Cloud Platform," in Proc. 2019 9th Int. Conf. on Cloud Computing, Data Science & Engineering (Confluence), Noida, India, 2019.

[2] A. B. Kuncara, D. S. Kusumo, and M. Adrian, "Comparison of Jenkins and GitLab CI/CD to Improve Delivery Time of Basu Dairy Farm Admin Website," Jurnal Teknik Informatika (Jutif), vol. 5, no. 3, pp. 747–756, 2024, doi: 10.52436/1.jutif.2024.5.3.1836.

[3] T. A. Ghaleb, O. Abduljalil, and S. Hassan, "CI/CD Configuration Practices in Open-Source Android Apps: An Empirical Study," ACM Trans. Softw. Eng. Methodol., 2025, doi: 10.1145/3736758.

[4] Rocket Lab, "How Long Does It Take to Build a Mobile App?" Rocket Lab Blog, n.d. [Online]. Available: https://www.rocketlab.com.au/blog/how-long-does-it-take-to-build-a-mobile-app

[5] JetBrains, "Kotlin Programming Language Reference," 2024. [Online]. Available: https://kotlinlang.org/docs

[6] Apple Inc., "Swift Programming Language Documentation," 2024. [Online]. Available: https://docs.swift.org

[7] Google, "Flutter Framework Documentation," 2024. [Online]. Available: https://docs.flutter.dev

[8] Meta Platforms, "React Native Documentation," 2024. [Online]. Available: https://reactnative.dev/docs

[9] Talance Co., Ltd. and Thai Programmer Association, "Thai Developer Salary Report 2023," Bangkok, Thailand, 2022.

[10] M. Nabil, "DevSecOps: Integrating Security into the DevOps Pipeline," J. Softw. Eng. Sec., vol. 14, no. 2, pp. 45–58, 2022.

[11] DORA, "Accelerate State of DevOps Report," DevOps Research and Assessment, 2022. [Online]. Available: https://dora.dev

[12] Speedscale, "Continuous Delivery Pipelines for Mobile Applications," Speedscale Documentation, n.d. [Online]. Available: https://speedscale.com/docs

[13] SonarSource, "SonarQube Documentation," 2023. [Online]. Available: https://docs.sonarsource.com

[14] OpenSecurity, "Mobile Security Framework (MobSF) Documentation," 2023. [Online]. Available: https://mobsf.github.io/docs/

[15] Fastlane, "Fastlane Documentation," 2024. [Online]. Available: https://docs.fastlane.tools

[16] Docker Inc., "Docker and Docker Compose Documentation," 2024. [Online]. Available: https://docs.docker.com

[17] PostgreSQL Global Development Group, "PostgreSQL 16 Documentation," 2024. [Online]. Available: https://www.postgresql.org/docs/16/

[18] J. W. Tukey, Exploratory Data Analysis. Reading, MA: Addison-Wesley, 1977.

[19] Kruskal, W. H., and Wallis, W. A., "Use of Ranks in One-Criterion Variance Analysis," J. Am. Stat. Assoc., vol. 47, no. 260, pp. 583–621, 1952.

[20] Dwass, M., "Some k-Sample Rank-Order Tests," in Contributions to Probability and Statistics, Stanford, CA: Stanford Univ. Press, 1960, pp. 198–202.